Deadlock analysis for open shop processing systems

Christian E.J. Eggermont[1], Alexander Schrijver[2], and Gerhard J. Woeginger[1]

[1] TU Eindhoven, Netherlands
[2] CWI and University of Amsterdam, Netherlands

Keywords: deadlock, resource allocation, computational complexity.

1 Introduction

We consider a *multi-stage open shop* processing system with n jobs $J_1, \ldots, J_n$ and m machines $M_1, \ldots, M_m$. Every job J_j ($j = 1, \ldots, n$) requests processing on a certain subset $\mathcal{M}(J_j)$ of the machines; the ordering in which job J_j passes through the machines in $\mathcal{M}(J_j)$ is irrelevant and can be chosen arbitrarily by the scheduler. Every machine M_i ($i = 1, \ldots, m$) has a corresponding *capacity* $\mathrm{cap}(M_i)$, which means that at any moment in time it can simultaneously hold and process up to $\mathrm{cap}(M_i)$ jobs. For more information on multi-stage scheduling systems, the reader is referred to the survey (LLRS 1993).

In this article, we are mainly interested in the performance of *real-time* multi-stage systems, where the processing time $p_{j,i}$ of job J_j on machine M_i is a priori unknown and hard to predict. The Central Control (the scheduling policy) of the system learns the processing time $p_{j,i}$ only when the processing of job J_j on machine M_i is completed. The various jobs move through the system in an unsynchronized fashion. Here is the standard behavior of a job in such a system:

1. In the beginning the job is asleep and is waiting outside the system. For technical reasons, we assume that the job occupies an artificial machine M_0 of unbounded capacity.
2. After a finite amount of time the job wakes up, and starts looking for an available machine M on which it still needs processing. If the job detects such a machine M, it requests permission from the Central Control to move to machine M. If no such machine is available or if the Central Control denies permission, the job falls asleep again (and returns to the beginning of Step 2).
3. If the job receives permission to move, it releases its current machine and starts processing on the new machine M. While the job is being processed and while the job is asleep, it continuously occupies machine M (and blocks one of the $\mathrm{cap}(M)$ available places on M). When the processing of the job on machine M is completed and in case the job still needs processing on another machine, it returns to Step 2.
4. As soon as the processing of the job on all relevant machines is completed, the job informs the Central Control that it is leaving the system. We assume that the job then moves to an artificial final machine M_{m+1} (with unbounded capacity), and disappears.

The described system behavior typically occurs in robotic cells and flexible manufacturing systems. The high level goal of the Central Control is to arrive at the situation where all the jobs have been completed and left the system. Other goals are of course to reach a high system throughput, and to avoid unnecessary waiting times of the jobs. However special care has to be taken to prevent the system from reaching situations of the following type:

Example 1. Consider an open shop system with three machines M_1, M_2, M_3 of capacity 1. There are three jobs that each require processing on all three machines. Suppose that the Central Control behaves as follows:

The first job requests permission to move to machine M_1. Permission granted.
The second job requests permission to move to machine M_2. Permission granted.
The third job requests permission to move to machine M_3. Permission granted.
Once the three jobs have completed their processing on theses machines, they keep blocking their machines and simultaneously keep waiting for the other machines to become idle. The processing never terminates.

Example 1 illustrates a so-called *deadlock*, that is, a situation in which the system gets stuck and comes to a halt since no further processing is possible: Every job in the system is waiting for resources that are blocked by other jobs that are also waiting in the system. Resolving a deadlock is usually expensive (with respect to time, energy, and resources), and harmfully diminishes the system performance. In robotic cells resolving a deadlock typically requires human interaction. The scientific literature on deadlocks is vast, and touches many different areas like flexible manufacturing, automated production, operating systems, Petri nets, network routing, etc.

The literature distinguishes two basic types of system states; see for instance Coffman, Elphick & Shoshani (CES 1971), Gold (Gold 1978), or Banaszak & Krogh (BK 1990). A state is called *safe*, if there is at least one possible way of completing all jobs. A state is called *unsafe*, if every possible continuation eventually will lead to a deadlock. An example for a safe state is the initial situation where all jobs are outside the system (note that the jobs could move sequentially through the system and complete). Another example for a safe state is the final situation where all jobs have been completed. An example for an unsafe state are the deadlock states.

2 The problems under investigation

In this article we will study safe and unsafe states in open shop scheduling systems. In particular, we will investigate the computational complexity of the following algorithmic questions. The most basic problem is to characterize the system states that can be reached while the shop is running.

Problem: Reachable State Recognition
Instance: An open shop scheduling system. A system state s.
Question: Can the system reach state s when starting from the initial situation?

If we want to have the system running smoothly, it is essential to distinguish safe from unsafe system states:

Problem: Safe State Recognition
Instance: An open shop scheduling system. A system state s.
Question: Is state s safe?

Another fundamental question is whether an open shop system can ever fall into a deadlock. In case it cannot, then there are no reachable unsafe states and the Central Control may permit all moves right away and without analyzing them; in other words the system is fool-proof and will also run smoothly without supervision.

Problem: Reachable Deadlock
Instance: An open shop scheduling system.
Question: Can the system ever reach a deadlock state when starting from the initial situation?

3 Basic definitions

A *state* of an open shop scheduling system is a snapshot describing a situation that might potentially occur while the system is running. A state s specifies for every job J_j

- the machine $M^s(J_j)$ on which this job is currently waiting or currently being processed,
- and the set $\mathcal{M}^s(J_j) \subseteq \mathcal{M}(J_j) - \{M^s(J_j)\}$ of machines on which the job still needs future processing.

The machines $M^s(J_j)$ implicitly determine

- the set $\mathcal{J}^s(M_i) \subseteq \{J_1, \ldots, J_n\}$ of jobs that are currently handled by machine M_i; since the machine capacities are to be respected we impose $|\mathcal{J}^s(M_i)| \leq \text{cap}(M_i)$.

The *initial state* 0 is the state where all jobs are still waiting for their first processing; in other words in the initial state all jobs J_j satisfy $M^0(J_j) = M_0$ and $\mathcal{M}^0(J_j) = \mathcal{M}(J_j)$. The *final state* f is the state where all jobs have been completed; in other words in the final state all jobs J_j satisfy $M^f(J_j) = M_{m+1}$ and $\mathcal{M}^f(J_j) = \emptyset$.

A state t is called a *successor* of a state s, if it results from s by moving a single job J_j from its current machine $M^s(J_j)$ to some new machine in set $\mathcal{M}^s(J_j)$, or by moving a job J_j with $\mathcal{M}^s(J_j) = \emptyset$ from its current machine to M_{m+1}. In this case we will also say that the system *moves* from s to t. This successor relation is denoted $s \to t$. A state t is said to be *reachable* from state s, if there exists a finite sequence $s = s_0, s_1, \ldots, s_k = t$ of states (with $k \geq 0$) such that $s_{i-1} \to s_i$ holds for $i = 1, \ldots, k$. A state s is called *reachable*, if it is reachable from the initial state 0. A state t is said to be *subset-reachable* from state s, if every job J_j satisfies one of the following three conditions:

- $M^t(J_j) = M^s(J_j)$ and $\mathcal{M}^t(J_j) = \mathcal{M}^s(J_j)$, or
- $M^t(J_j) \in \mathcal{M}^s(J_j)$ and $\mathcal{M}^t(J_j) \subseteq \mathcal{M}^s(J_j) - \{M^t(J_j)\}$, or
- $M^t(J_j) = M_{m+1}$ and $\mathcal{M}^t(J_j) = \emptyset$.

Clearly whenever a state t is reachable from some state s, then t is also subset-reachable from s. The following example demonstrates that the reverse implication is not necessarily true. This example also indicates that the algorithmic problem Reachable State Recognition (as formulated in the introduction) is not completely straightforward.

Example 2. Consider an open shop system with two machines M_1, M_2 of capacity 1 and two jobs J_1, J_2 with $\mathcal{M}(J_1) = \mathcal{M}(J_2) = \{M_1, M_2\}$. Consider the state s where J_1 is being processed on M_1 and J_2 is being processed on M_2, and where $\mathcal{M}^s(J_1) = \mathcal{M}^s(J_2) = \emptyset$. It can be seen that s is subset-reachable from the initial state 0, whereas s is not reachable from 0.

A state is called *safe*, if the final state f is reachable from it; otherwise the state is called *unsafe*. A state is a *deadlock*, if it has no successor states and if it is not the final state f. A state is called *fool-proof*, if no deadlock states are reachable from it.

4 Summary of results

Consider a fixed system state s. A machine M is *full* in state s, if it is handling exactly $\text{cap}(M)$ jobs. A non-empty subset $\mathcal{B}$ of the machines is called *blocking* for state s, if

- every machine in $\mathcal{B}$ is full, and if
- every job J_j that occupies some machine in $\mathcal{B}$ satisfies $\emptyset \neq \mathcal{M}_s(J_j) \subseteq \mathcal{B}$.

In other words, the machines in $\mathcal{B}$ all operate at full capacity on jobs that in the future only want to move to other machines in $\mathcal{B}$. Since these jobs are permanently blocked from moving, the stage s must eventually lead to a deadlock; hence s is unsafe. The following (well-known) theorem demonstrates that also the reverse statement holds.

Theorem 1. *A state s is an unsafe state, if and only if s has a blocking set of machines. Furthermore, it can be decided in polynomial time whether s has a blocking set of machines.*

This implies a polynomial time algorithm for problem SAFE STATE RECOGNITION. There is a fairly simple way of rewriting the REACHABLE STATE RECOGNITION problem into an instance of SAFE STATE RECOGNITION (by essentially reversing the direction of time). This then yields the following theorem.

Theorem 2. *Problem* REACHABLE STATE RECOGNITION *can be decided in polynomial time.*

In strong contrast to these positive results, we show that problem REACHABLE DEADLOCK is intractable. This is done by a reduction from the NP-hard three-dimensional matching problem.

Theorem 3. *Problem* REACHABLE DEADLOCK *is NP-complete, even if the capacity of each machine is at most three, and if each job needs processing on at most four machines.*

Finally we analyze two tractable special cases of REACHABLE DEADLOCK. In both cases we translate the problem into a graph-theoretic setting. The solution of the first tractable case uses a convex programming formulation and techniques from matching theory. The solution of the second tractable case analyzes cycles in certain edge-colored graphs.

Theorem 4. *If each job requires processing on at most two machines, then* REACHABLE DEADLOCK *can be solved in polynomial time.*

Theorem 5. *If each machine has capacity one, then problem* REACHABLE DEADLOCK *can be solved in polynomial time.*

References

Banaszak, Z.A., and B.H. Krogh, 1990, "Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows", *IEEE Transactions on Robotics and Automation* 6, pp. 724-734.

Coffman, E.G., M.J. Elphick, and A. Shoshani, 1971, "System Deadlocks", *ACM Computing Surveys* 3, pp. 67-78.

Gold, M. 1978, "Deadlock prediction: Easy and difficult cases", *SIAM Journal on Computing* 7, pp. 320-336.

Lawler, E.L., J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, 1993, "Sequencing and scheduling: Algorithms and complexity", *Handbooks in Operations Research and Management Science*, Vol. 4, North Holland, pp. 445-522.